Billing7

Work in: https://github.com/wwprovost/<your-team>

In this workshop your team will build out a set of HTTP interfaces to the Billing system, and add an auditing feature. You'll develop under continuous integration, using a provided Jenkins server.

Instructions for this workshop are less detailed, and you'll be challenged to develop your own solutions to given requirements, with some guidance along the way and prior code examples at hand.

The starter code is found in **Billing6**, and will also be loaded into a Git repository that's been set up for this project. It is more or less the answer code to the previous workshop, using MongoDB as the database, and with a couple of tweaks to simplify further work:

- The **Customer** class is reworked so that a **name** field is now the primary key for the class. The **_id** field suggested in the previous set of instructions would be carried into HTTP and JSON representations; and the full customer name really has been our primary key all along. So, though it's a bit brittle, the approach now is to forge the **name** from first and last names given to the constructor, and to report them via getters for code already in the application, relying on parsing the name by splitting on the space character. (So apparently our business can't have Jan Michael Vincent as a customer – alas.)
- The **MongoPersistence** class is a Spring **@Primary @Component**. This means that the main or "normal" configuration of one **Reporter**, one **Billing** object, and one **Persistence** component (**MongoPersistence** being chosen to satisfy this last dependency) is the easiest one to configure. The **ParserPersistence** class is an ordinary **@Component**, and can be autowired by its specific type, as is done in the **Migration** component.
- Many constants, helper methods, and custom matchers have been refactored to the **TestUtility** class, so there's less code and less repetition in the unit and integration tests.

Requirements

Here are the high-level goals, roughly in order of priority. Complete as many of these requirements as you can in the available time.

- Build a new **BillingRS** Spring Boot application that supports multiple REST controller components.
- Support the existing functionality of the **Billing** and **Reporter** components by way of HTTP requests. This includes retrieving customers and invoices by ID and all of the existing query methods on **Billing** (though a complete set of CRUD operations is not desired); the three update methods on **Billing**; and being able to retrieve the latest reports as created by the **Reporter** component.
- Make the application track all changes to customers and invoices, recording the date and time of the change and the complete state of the new or changed object.
- Provide an HTTP interface to this new feature, as well: callers should be able to read a log of all updates, or all updates since a given date and time.
- Solid unit testing and integration testing is understood to be a part of all of the above features. Your Jenkins project should rebuild on any code push, all unit tests should be passing, integration tests should be excluded, and the number of (meaningful!) tests and amount of code under tests should gradually increase.
- Functional tests for the running service would be nice, too!
- A refinement of the audit feature might be to limit the update records to the state that changes, rather than deep copies of the whole modified customer or invoice.

Process

First, working as a group, you'll do some design and up-and-running work:

- Design the HTTP APIs: what HTTP methods, URLs, parameters, and content types will be involved in driving the Billing system? Sketch out the specific HTTP requests and identify what responses will contain; also consider likely error conditions and identify HTTP response codes for them. Review the API with the instructor.
- Design an object-oriented and Spring-friendly solution for the auditing feature: what classes will you need, and of what stereotypes – that is, domain objects, persistence services, controllers, etc.? Review this design with the instructor.
- Create your Jenkins project and get it to the point of running an initial, successful build.
- Create the **BillingRS** class, and a starter version of one of the REST controllers, and get it up and running, to the point at which you can make an HTTP invocation and get a clean response. This doesn't have to involve any domain classes or the database yet; a simple "Hello, web service" interaction is fine.
- Push this code to your Git repository so everyone has a base version of the application.

Then you can start to tackle the various controllers, domain classes, persistence logic, unit tests, and integration tests individually. Divide the work however makes sense to you. We're not going for any strict methodology here but try to keep Agile and CI practices in mind. Especially, try to work in small, testable steps, even if it's just a method at a time, and be checking code in frequently. It's a bad sign if a lot of new functionality is only working on one person's system or only on one Git branch! Keep communicating, and remember that the smaller and more discrete are your development tasks, the more able you will be to adapt as a team, balance the load, and adjust to unexpected challenges.